Assignment 3

Team number: 2
Team members

Name	**Student Nr.**	**Email**
Lucas Ponticelli	2687740	l.p.i.ponticelli@student.vu.nl
Khagan Mammadov	2669723	k.mammadov@student.vu.nl
Jeroen Klaver	2696798	j.a2.klaver@student.vu.nl
David Breitling	2685269	d.k.l.breitling@student.vu.nl

Format: Class, package and library names are **bold**; attributes, operations, and associations are underlined text; objects are *italic*.

Summary of changes of Assignment 2

Author(s): David Breitling, Khagan Mammadov

Most of the feedback we received on our second assignment was very positive, so we did not have to revise much.

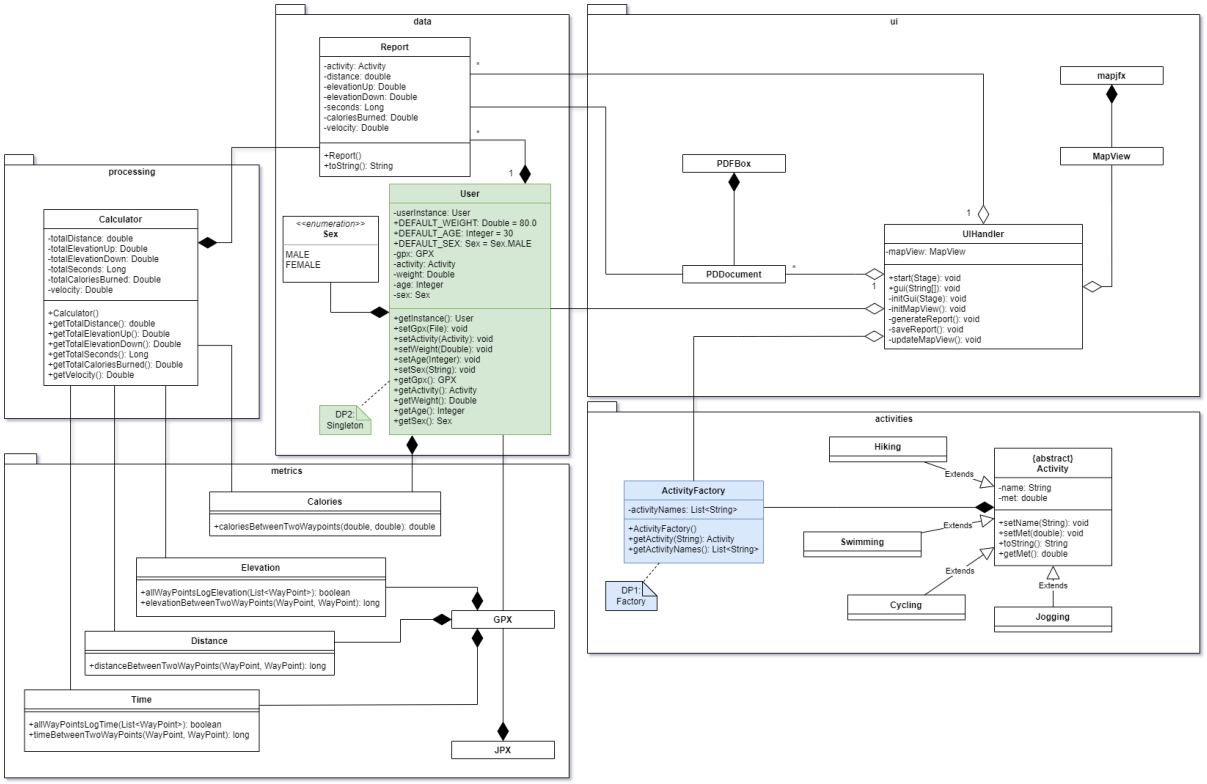
- The biggest issue pointed out by our TA was that our state machine diagrams, especially the GUI state machine diagram, were “actually activity diagrams, and not state machine diagrams per se”. We reworked these diagrams to accurately model the states of the entities and their transitions.

- With regards to the class and sequence diagrams, the feedback we got was very positive. For both, the only suggestion for improvement was to include some alternative approaches that we could have taken in the description of our diagrams, which we made sure to include in our revised version.

- For the object diagram, we had not represented the **MapVisualiser** class although we should have. Since we ended up taking a different approach to actually displaying the map, this issue has been fixed on its own for this assignment. However, we made sure to include it in our revised version.

Application of design patterns

Author(s): Khagan Mammadov



	DP1
Design pattern	Factory
Problem	The main technical challenge of our application has been to make it easily extensible with different types of activities. We solved this problem through the implementation of the extensible abstract class **Activity**. However, we soon realised that the addition of a new activity to our application involved an extra step: every activity must also be added to a list accessible to the **UIHandler** for it to be displayed as an option to the user. Our goal was to automate this extra step to make our application as easily extensible as possible.
Solution	Our solution to this problem was to dynamically parse every class name in the **activities** package and store them in a list upon the creation of an **ActivityFactory** object. This list of activity names is then later retrieved by the **UIHandler** class to be displayed to the user. **ActivityFactory** follows the standard Factory design pattern and instantiates objects based on their class name. We chose to automate this process by using the

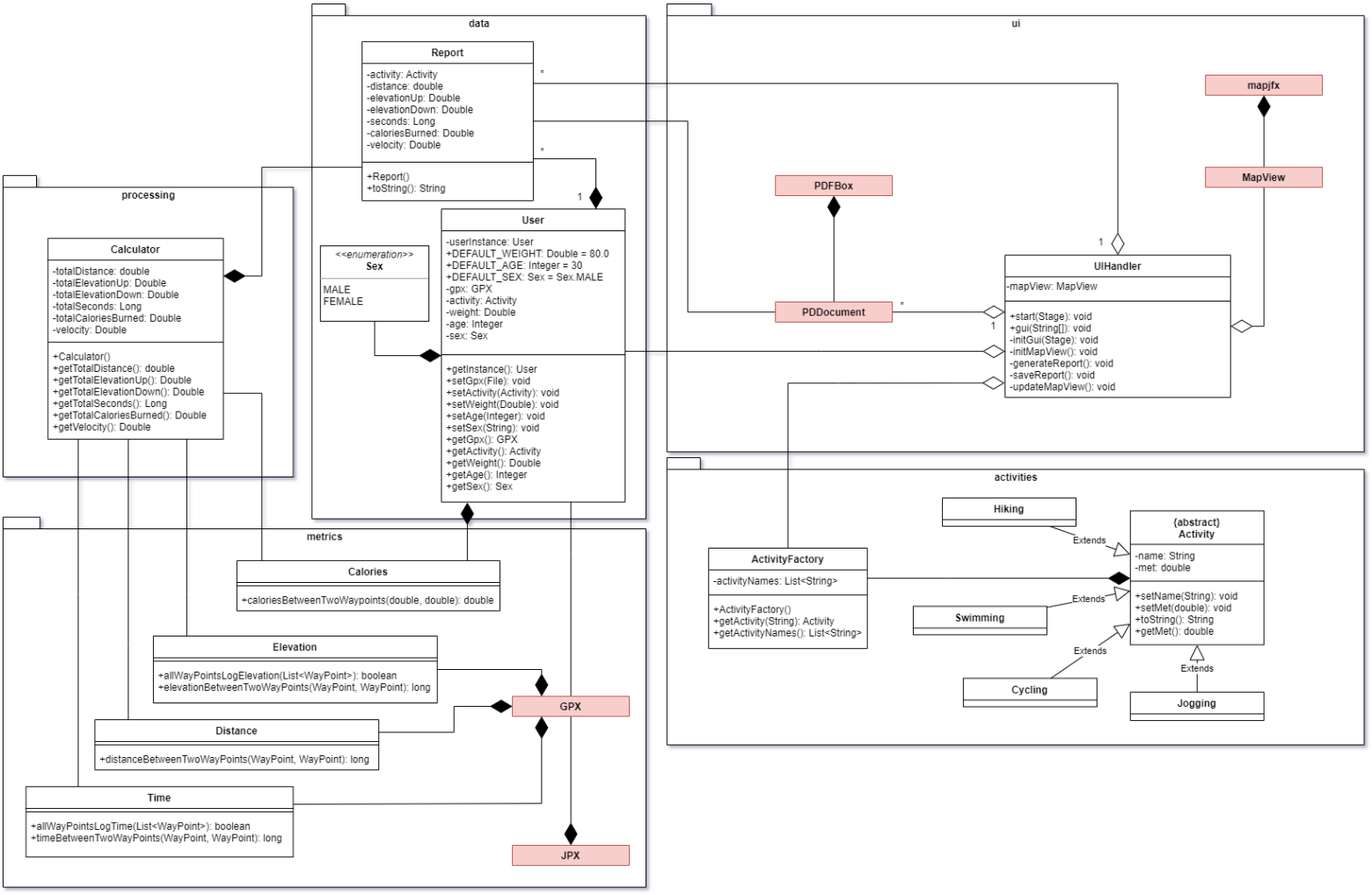
	newInstance() method of the **Class** class to instantiate classes in the **activities** package given their name. An alternative to this would be to use a switch statement with static values, which would fail to solve our problem. The combination of a dynamically parsed list of class names and the newInstance() method of the **Class** class eliminates the need for developers to modify a switch statement when they add new activities.
Intended use	Upon the initialisation of the activity **ChoiceBox**, an **ActivityFactory** object is created for the **UIHandler** class to retrieve all the activity names in the **activities** package. The getActivityNames() method of **ActivityFactory** is used to retrieve the list of activity names. The activity **ChoiceBox** is then initialised with the list retrieved from the **ActivityFactory** object. This list of activity names is then displayed for the user to choose from. Once an activity name is chosen, the getActivity() method of **ActivityFactory** is called to instantiate the activity with that name. An alternative to this would be to display a list of instantiated activities instead, however this method would fail to fully utilise the **ActivityFactory** class and scale poorly with a large number of activities.
Constraints	The only constraint imposed by our application of the Factory design pattern is our implementation of it. As we dynamically parse the **activities** package, we have to manually implement rules to avoid instantiating classes such as **Activity** and **ActivityFactory** when creating the list of activities available to the user. As such, if a class that is not intended to be presented to the user as an activity is added to the **activities** package, a rule preventing this must also be added to **ActivityFactory**.

	DP2
Design pattern	Singleton
Problem	We initially based our implementation of the **User** class around a general procedure: a new **User** instance would be created upon launch and the creating class, **UIHandler**, would keep the reference to the object and pass it as an argument to other classes. We later realised that our application currently serves a single user at a time and the possibility of multiple **User** instances existing simultaneously implies otherwise.
Solution	We modified our **User** class to follow the Singleton design pattern by making the constructor private and adding a getInstance() method. The Singleton **User** class clearly conveys the existence of a single **User** instance throughout the execution of our application. The existence of a single **User** instance also eliminates the need to explicitly instantiate a **User** object in **UIHandler** and pass it as an argument to other classes.
Intended use	As there is effectively a single global **User** instance every class can refer to, the **UIHandler** class does not need to explicitly instantiate a **User** object at launch. Ideally, the first getInstance() call is still made by the **UIHandler** class to set the .gpx file of the user. The Singleton **User** instance is then utilised by the **UIHandler** class to control GUI behaviour and set the attributes of the instance to values taken from the user as

	input. The Singleton **User** instance is then utilised by the **Report** class, the **Calculator** class and the **metrics** package to generate a report on demand.
Constraints	An obvious constraint imposed by the application of the Singleton design pattern is that any potential functionalities that would require more simultaneous users are limited to a single user instead. This is not a concern at this stage of application as it is limited to a single user, but could prove to be an obstacle should any such functionalities be implemented in the future.

Class diagram

Author(s): Lucas Ponticelli, Jeroen Klaver



All of the relationship multiplicities are assumed to be 1-to-1 if not specified in the above class diagram. External classes and libraries are highlighted in salmon.

Package: data

Report

Report is a class that contains the different attributes that will be displayed in the summary report for the user to then save or just view. It has a composition relationship with the **Calculator** and **User** class as a report may not exist without both of these classes. It also has a bi-directional relationship with the **PDDocument** and a shared aggregation with the **UIHandler** as it is called by pressing a button in the UI of our application.

Attributes

- caloriesBurned: Double – Attribute to save the total amount of calories burned by the user
- elevationUp: Double – Attribute to save the total positive elevation of the workout
- elevationDown: Double – Attribute to save the total negative elevation of the workout

Operations

- toString(): String – Operation to store all the attributes of a **Report** object in a single string. Utilised by the **UIHandler** methods generateReport() and saveReport() to pretty-print the generated report

User

User is a class with 9 attributes and 11 getter and setter operations. This class has a shared aggregation with the **UIHandler** class. It is used to store the chosen .gpx file, activity and biometrics of the user to be later used by the **Calculator**.

Attributes

- DEFAULT_WEIGHT: Double – Default value of the weight if the user does not wish to input their weight to be later used in the calculations
- DEFAULT_AGE: Integer – Default value of the age if the user does not wish to input their age to be later used in the calculations
- DEFAULT_SEX: Sex – Default value of the sex if the user does not wish to input their sex to be later used in the calculations

<<enumeration>> Sex

Sex has a composition relationship with the **User**. We chose to use an enumeration here as it more explicitly represents the sex of the user than a boolean.

Package: processing

Calculator

Calculator is a class in charge of adding up the results from each class that is connected to it from the **metrics** package to get the final result. These different classes have operations that deal with calculating the different parameters between 2 waypoints. It has 4 bi-directional relationships with classes in the **metrics** package namely **Calories**, **Elevation**, **Distance** and **Time**. All of these relationships are required to calculate the different values to then use for the calculation of calories. This implementation allowed for a modular way of adding new metrics to the **metrics** package. An alternative to this would be the method we previously used in Assignment 2 which involved calculating all the metrics within **Calculator**. We decided against this as creating explicit classes for each metric results in cleaner code and a more convenient way to extend the **metrics** package with new types of metrics. It has a constructor and 6 getters to be able to access its values in different classes.

Attributes

- totalCaloriesBurnt: Double – Attribute to save the calories burnt between waypoints and then added up to get the total calories burnt in their workout
- Velocity: Double – Attribute to save the average velocity throughout the workout calculated by dividing totalDistance by totalSeconds

Package: metrics

GPX

GPX is a class imported from the external **JPX** library to aid with parsing and performing calculations on a chosen .gpx file with the help of 3 other classes: **Distance**, **Elevation** and **Time**. They all have a composition relationship with the **GPX** class as these classes cannot exist without a .gpx file. It also has a composition relationship with the external **JPX** library.

Time

Time is a class with 2 static methods that will take care of calculating all time related metrics. It has a composition relationship with the **GPX** class as it calculates the time spent between each waypoint.

Operations

- allWayPointsLogTime(List<WayPoint>): boolean – Operation to check if all the waypoints in a .gpx file have a time log
- timeBetweenTwoWayPoints(WayPoint, Waypoint): long – Operation to get the time difference between 2 waypoints in a .gpx file

Calories

Calories is a class with a single static method to calculate the calories burnt between 2 waypoints, caloriesBetweenTwoWaypoints. It has a composition relationship with **User** as it uses the biometrics from the user to accurately calculate the calories burnt.

Package: activities

<abstract> Activity

Activity is an abstract class with 2 attributes, and 4 operations. It is extended currently by 4 other classes: **Cycling**, **Jogging**, **Swimming** and **Hiking**. It represents the different activities that are currently included with our application and is easily extensible. We chose to make this class abstract to provide developers with the attributes and methods required to conveniently extend our application with new activities. An alternative to this would be to use an interface, however we found that our application could not utilise the flexibility of implementing methods differently for each class.

Attributes

- name: String – The name of the activity that is assigned in the extending classes of **Activity**
- met: double – The metabolic equivalent of a task used to accurately calculate the amount of calories used in different sports

ActivityFactory

ActivityFactory is a class used to create activity objects given their name without hardcoding their type in the code. It thus has a composition relationship with **Activity**, as **ActivityFactory** cannot exist without it. It also has a shared aggregation with the **UIHandler** as it is instantiated in the initActivityChoiceBox() method called by **UIHandler** when initialising the GUI.

Attributes

- activityNames: List<String> – Attribute to store the name of all the activities in the **activities** package

Package: ui

MapView

MapView is a class used to visualise .gpx files and is imported from the external **mapjfx** library shown in our class diagram. It has a composition relationship with **mapjfx** as it is part of it. It also has a shared aggregation with the **UIHandler** as it is instantiated by the GUI of our app once the initMapView() method is called.

PDDocument

PDDocument is a class used to create PDF files and is imported from the external **PDFBox** library seen in our class diagram. It has a composition relationship with **PDFBox** as it is part of it. It also has a shared aggregation with the **UIHandler** as it is instantiated by the GUI of our app once the saveReport() method is called.

UIHandler

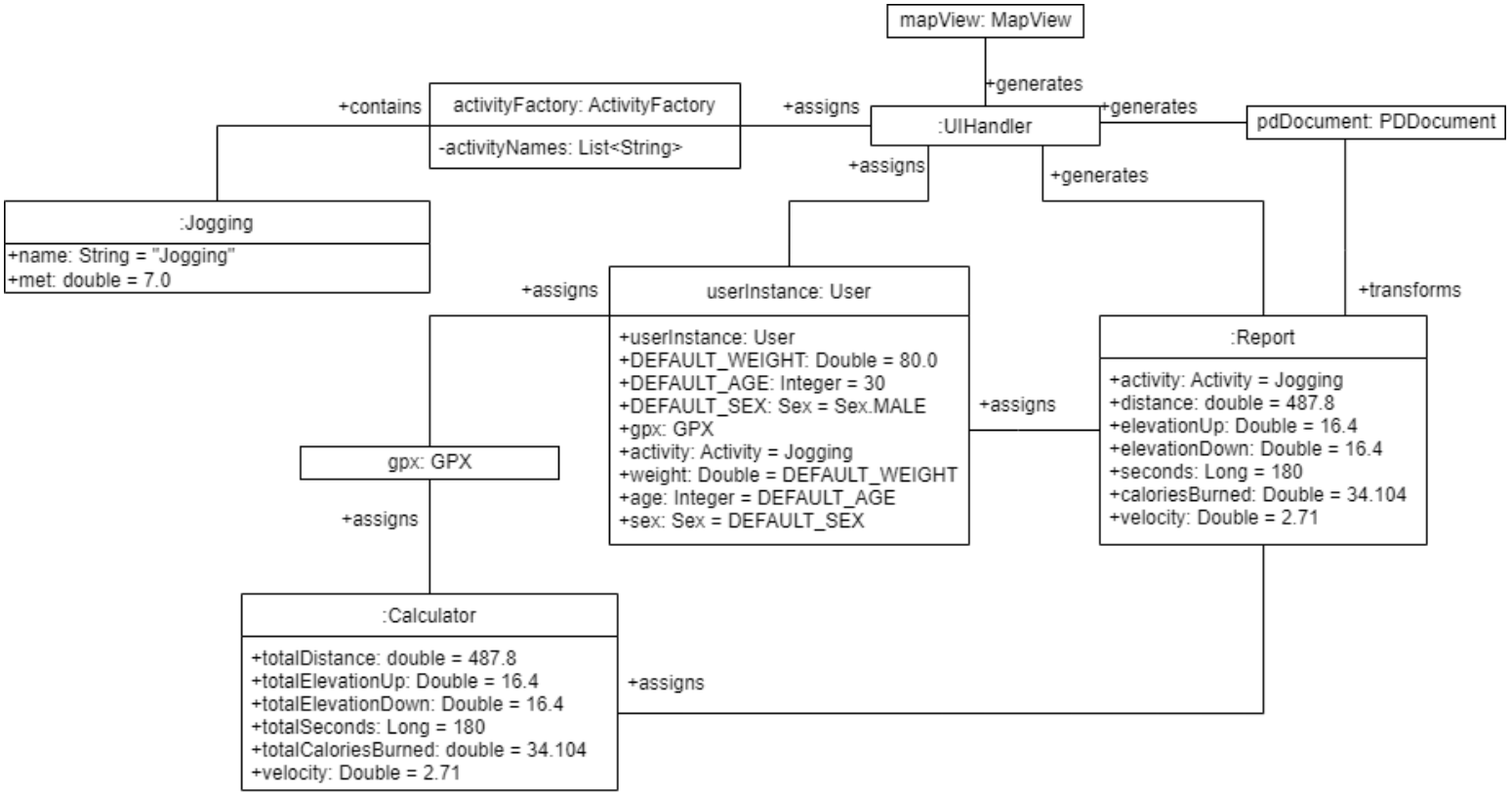
UIHandler is a class used to handle the GUI and its interactions with the other classes in our application. All of the relationships of **UIHandler** have already been stated by the classes described above. An alternative to our implementation of **UIHandler** would have been to abstract its interactions with other classes away into a separate class that would act as an intermediary between **UIHandler** and the backend.

Operations

- initGui(Stage): void – Operation that initiates the whole graphical interface of our application when the app is first launched
- initMapView(): void – Operation to initialise the **MapView** object to be later updated with the different coordinates from the .gpx file through the updateMapView() method
- generateReport(): void – Operation that creates a new **Report** which will be populated with metrics from the calculator
- saveReport(): void – Operation that first takes a snapshot of the **MapView** object and then pretty-prints the **Report** into a PDF document chosen by the user
- updateMapView(): void – Operation that updates the **MapView** object with the coordinates from the chosen .gpx file

Object diagram

Author(s): Jeroen Klaver, Lucas Ponticelli



This object diagram displays a “snapshot” of the system during which the system has finished creating the report. All of the input variables have already been set by this time, the calculations have been finished and the report has obtained the necessary values.

- **ActivityFactory** and **Jogging** - In this instance the user has selected the **Jogging** activity which assigns specific metrics to be used by the **Calculator** class later. These values will be different for each activity. The **ActivityFactory** meanwhile allows for the dynamic creation of activities, further detailed in our design pattern section under ‘Factory’ (DP1).

- **User** - This class contains the biometric data that can be filled in by the user, as well as the activity and .gpx file that have been selected. If the user does not fill in their personal data then default values will be used instead. We can see from the values inside of the class that the user in this instance has elected not to provide biometric information and that the default values are used instead. An alternative to this could have been to have a default User class that gets used when the user does not input biometrics. However, our current implementation allows us to use the Singleton design pattern which is further detailed in our design pattern section under ‘Singleton’ (DP2).

- **GPX** and **Calculator** - The .gpx file read into the **GPX** object contains a stream of waypoints that contain the metrics already displayed in the diagram. We then loop over each waypoint while calling static methods from classes that are not instantiated and therefore not displayed on this object diagram. Initially these classes, namely **Time**, **Distance** and **Elevation** were instantiated but we never operated on these classes and simply used their methods to get the desired metrics. The values obtained are then sent towards the **Calculator** which obtains all its metrics from the **User** and **GPX** classes to calculate values that can then be displayed in the **Report** class.

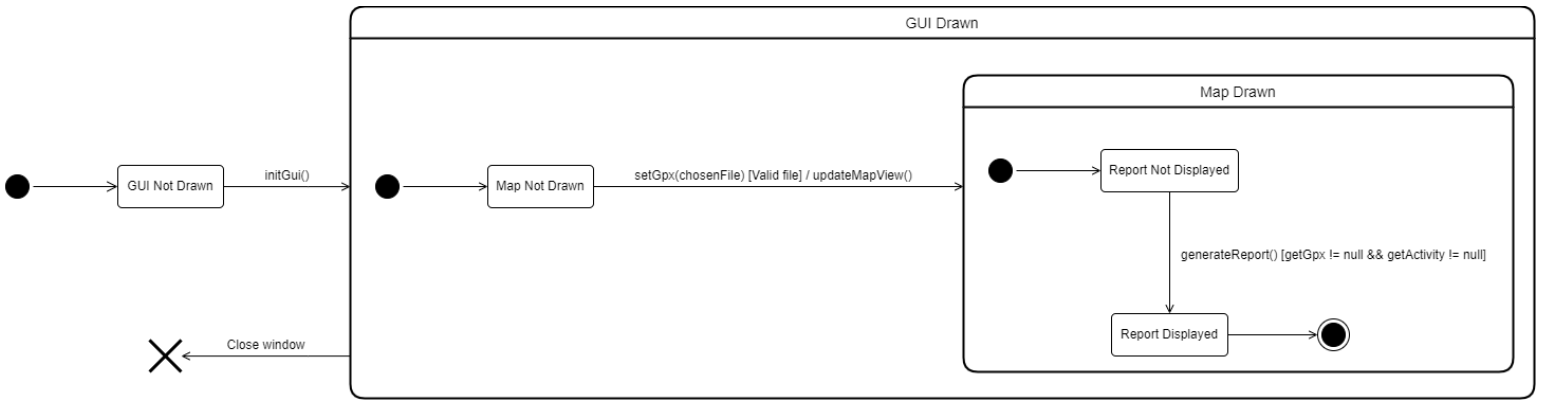
- **Report** and **PDDocument** - The **Report** class uses the calculated values from the **Calculator** to initialise its own attributes. **PDDocument** is part of the external **PDFBox** library used to save the report as a PDF.

- **UIHandler** and **MapView** - The **UIHandler** class is not instantiated but connects all of the classes in our application. We included it in the object diagram as it is fundamental to understanding our system. It is responsible for setting user biometrics, setting an activity, setting the .gpx file and generating and saving reports. **MapView** is a part of the external **mapjfx** library that is used to accurately visualise and display on screen the chosen .gpx file.

State machine diagrams

Author(s): Khagan Mammadov, David Breitling

State Machine Diagram - GUI



See above for a state machine diagram of the **UIHandler** class. The **UIHandler** class is responsible for controlling the interaction of the user with the system and the GUI. Initially, as the application is launched, the *GUI Not Drawn* state is entered. Once the initGui() method is called, the GUI is initialised and the *GUI Drawn* state is entered.

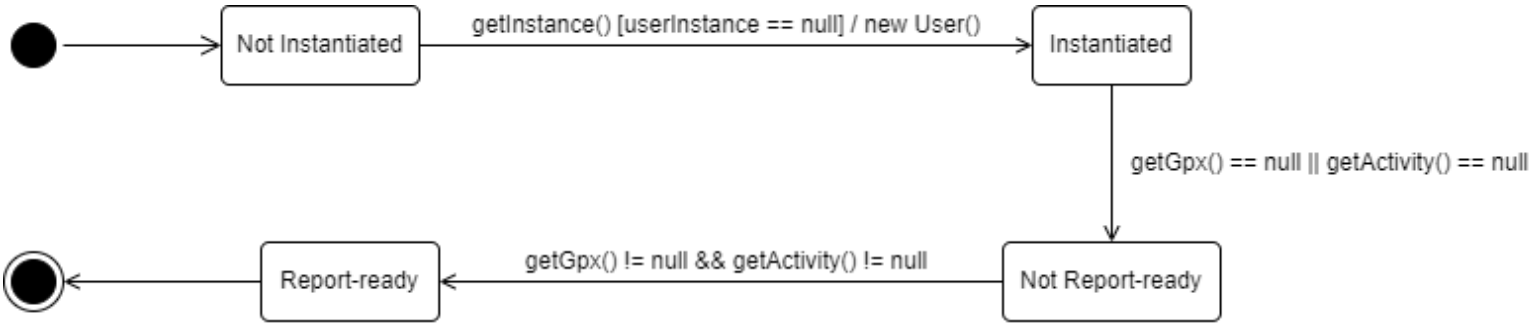
Initially, the *Map Not Drawn* substate is entered as the user has not yet chosen a .gpx file. The transition to the *Map Drawn* state is triggered once a valid .gpx file is chosen. This transition also triggers the updateMapView() method which updates the **MapView** object with data from the chosen .gpx file. Choosing an invalid file causes the GUI to remain in the *Map Not Drawn* state until a valid .gpx file is chosen.

Once the *Map Drawn* substate is entered, the user is presented with a visualisation of the chosen .gpx file. Initially, the machine remains in the *Report Not Displayed* substate. The transition to the *Report Displayed* substate is triggered once the user chooses to generate a report given that they have selected a valid file and activity. The **TextField** object displayed on screen is updated once the generateReport() method is triggered. An alternative to this would be to automatically generate and display a report without the user explicitly having to press the ‘Generate report’ button. The condition for a valid file and activity would then be checked once a file or activity is chosen. However, we decided to include a button to generate a report, in order to provide a more tactile experience.

Once the *Report Displayed* substate is entered, the user may choose to remain in this state, close the application, or choose a new .gpx file. If the user chooses a new .gpx file, the state of the machine will not be affected by the validity of the file. This is because an invalid file causes the **MapView** object to not get updated and keep displaying the visualisation of the previous .gpx file. Similarly, not updating the **TextField** object on screen causes the GUI to keep displaying the report of the previous .gpx file. *Map Drawn* and *Report Displayed* ultimately remain the final substates of the machine as the displayed visualisation and report

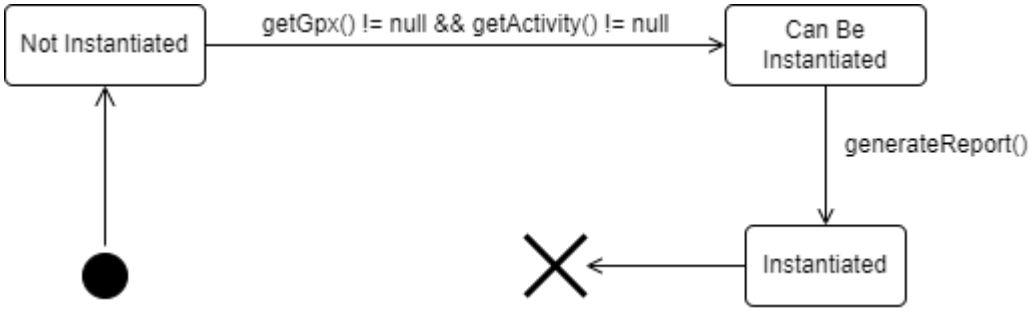
are never cleared. An alternative to this would be to reset the **MapView** and **TextField** objects if an invalid file is chosen, however we decided against it as we felt it was redundant.

State Machine Diagram - User



See above for the state machine diagram of the **User** class. From the initial state, the **User** class enters the *Not instantiated* state. In the event that getInstance() is called, the state transitions to the *Instantiated* state. During the transition, if userInstance is `null`, `new User()` is executed. In the event that either getGpx() or getActivity() return `null`, the *Not Report-ready* state is entered. In the event that both getGpx() and getActivity() both do not return `null`, the final *Report-ready* state is entered. Note that getGpx() and getActivity() are **User** methods. In the actual implementation, they are being called as User.getInstance().getGpx() and User.getInstance().getActivity() respectively. An alternative to this would be to instantiate **User** objects on demand, however this would falsely imply the possibility of the simultaneous existence of several **User** objects.

State Machine Diagram - Report

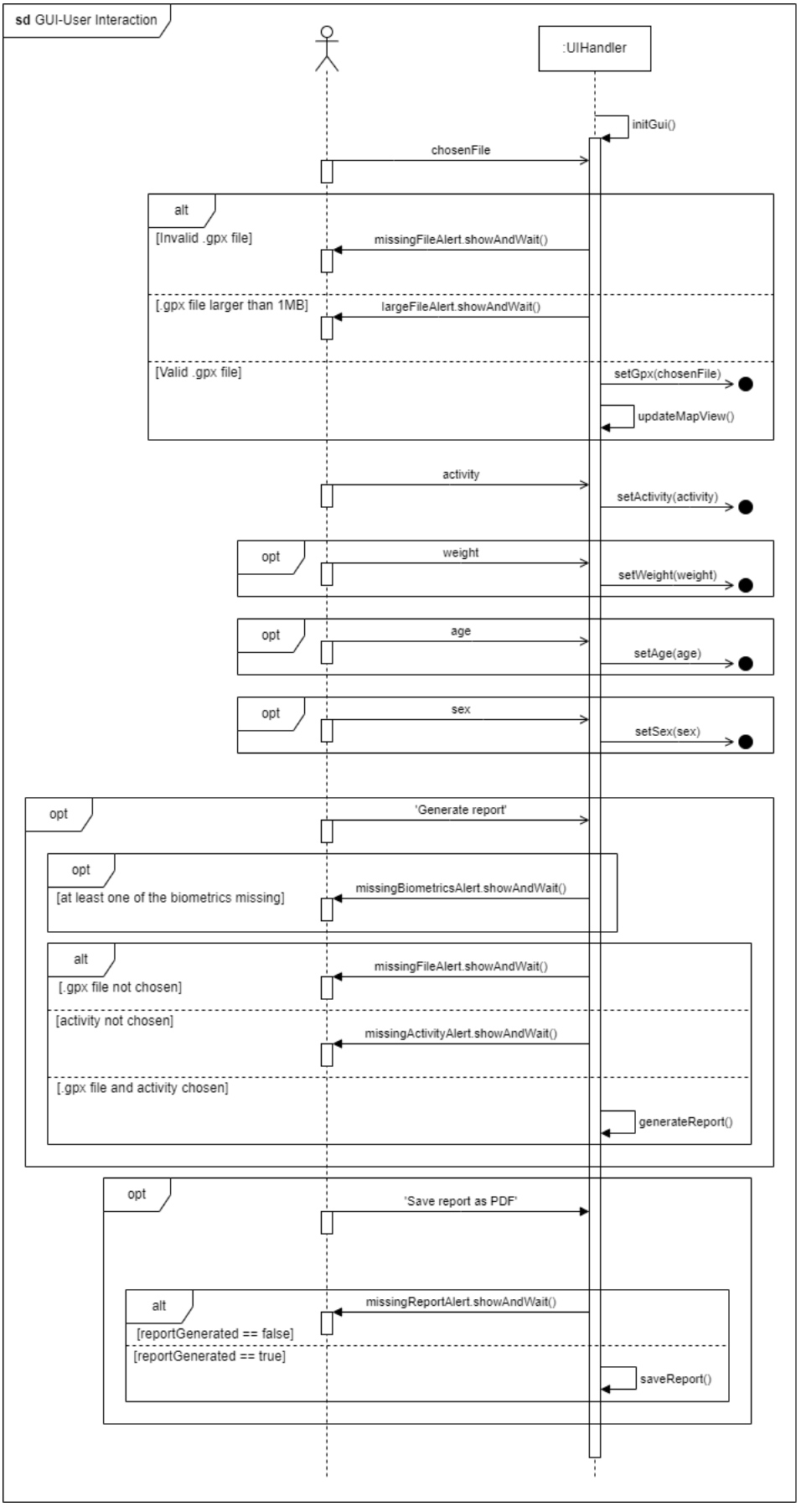


See above for the state machine diagram of the **Report** class. From the initial state, the **Report** class enters the *Not Instantiated* state. In the event that both getGpx() and getActivity() do not return `null`, the *Can Be Instantiated* state is entered. In the event that generateReport() is called, this state transitions to the *Instantiated* state, from where the instance of **Report** is destroyed. An alternative to this would be to apply the Singleton design pattern to the **Report** class and to set its parameters when needed instead of instantiating a new **Report** object on demand, however we found this to bloat our code with redundant setter operations.

Sequence diagrams

Authors: Khagan Mammadov, David Breitling

Sequence Diagram - GUI-User Interaction



The diagram above details how the user interacts with the GUI of our system. Initially, the initGui() method is called by **UIHandler** to initialise the GUI. Once the user chooses a .gpx file, one of three alternative interactions can occur:

- If the user chose an invalid .gpx file, missingFileAlert.showAndWait() is called and an alert is displayed on screen
- If the user chose a .gpx file larger than 1MB, largeFileAlert.showAndWait() is called and an alert is displayed on screen
- If the user chose a valid .gpx file, setGpx(chosenFile) is called to set the .gpx file of the user and then updateMapView() is called to update the **MapView** object with coordinates from the .gpx file

Once the user chooses an activity, setActivity(activity) is called to set the activity of the user. The user can then optionally insert biometric data such as weight, age and sex which will then cause for the methods setWeight(weight), setAge(age) and setSex(sex) to be called respectively.

The user can then optionally choose to generate a report. If the user has not provided at least one of the biometrics mentioned above, missingBiometricsAlert.showAndWait() is called and an alert is displayed on the screen. Then, one of three alternative interactions can occur:

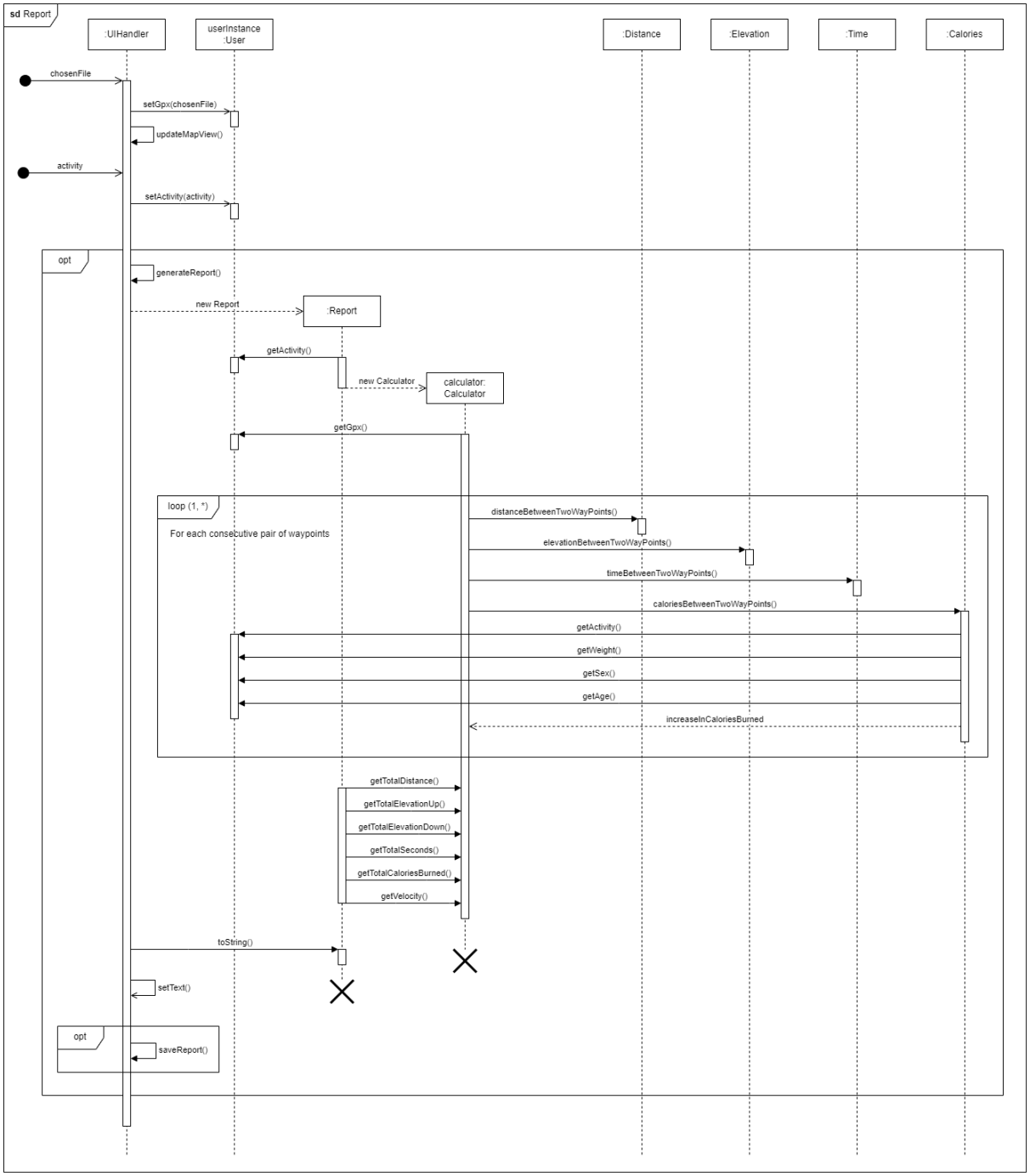
- If the user has not chosen a .gpx file, missingFileAlert.showAndWait() is called and an alert is displayed on screen
- If the user has not chosen an activity, missingActivityAlert.showAndWait() is called and an alert is displayed on screen
- If the user has chosen both a .gpx file and an activity, generateReport() is called and the generated report is displayed on screen

The user can then optionally choose to save the generated report. Then, one of two alternative interactions can occur:

- If the user has not generated a report, missingReportAlert().showAndWait() is called and an alert is displayed on screen
- If the user has generated a report, saveReport() is called and the generated report is saved as a PDF file

An alternative to this would be to save the generated report in a simpler format, however we wanted to include a snapshot of the **MapView** object in the PDF file.

Sequence Diagram - Report Creation



The diagram above details how the **Report** class functions in our system. setGpx(chosenFile) is called once the user has selected a .gpx file (see **SD** GUI-User Interaction). The controlling instance (of **UIHandler**) then gets the path to the .gpx file. Upon a successful read operation, the **UIHandler** instance does a self-call to updateMapView() to update the **MapView** object with coordinates from the .gpx file and visualise the map. Once this has completed the user can now set the activity which is required in order for the **Calculator** class to calculate the necessary metrics before creating a report. Optionally, if the user chooses to generate a report, the **UIHandler** instance self-calls generateReport(),

which instantiates a new **Report**. This **Report** instance in turn instantiates a new **Calculator**, and calls getActivity() to obtain the previously set activity. Just before starting its calculations, it calls getGpx() to get the necessary GPX information from the **User** object. Afterwards, *calculator* executes calls to **Distance**'s static distanceBetweenTwoWayPoints() method, **Elevation**'s static elevationBetweenTwoWayPoints() method, **Time**'s static timeBetweenTwoWayPoints() and **Calories**' static caloriesBetweenTwoWayPoints() method for each consecutive pair of waypoints. Before the Calculator is able to calculate and return the amount of calories burned, it has to gather the biometric information from the **User** object through the getter methods getActivity(), getWeight(), getSex() and getAge(). After having calculated all these metrics for every waypoint, the **Report** object calls *calculator*'s getTotalDistance(), getTotalElevationUp(), getTotalElevationDown(), getTotalSeconds(), getTotalCaloriesBurned(), and getVelocity() methods. Subsequently, the **Report** object is returned to the **UIHandler** instance, which then executes a call to toString(), used to pretty-print the report, as well as setText(), which sets the text of the **TextField** object on screen. An alternative to this would be to display the generated report in a separate window instead, however we decided against this as we wanted all of the relevant information to be displayed on the same window. Optionally, the **UIHandler** does a self-call to saveReport() if the user chooses to save the report as a PDF.

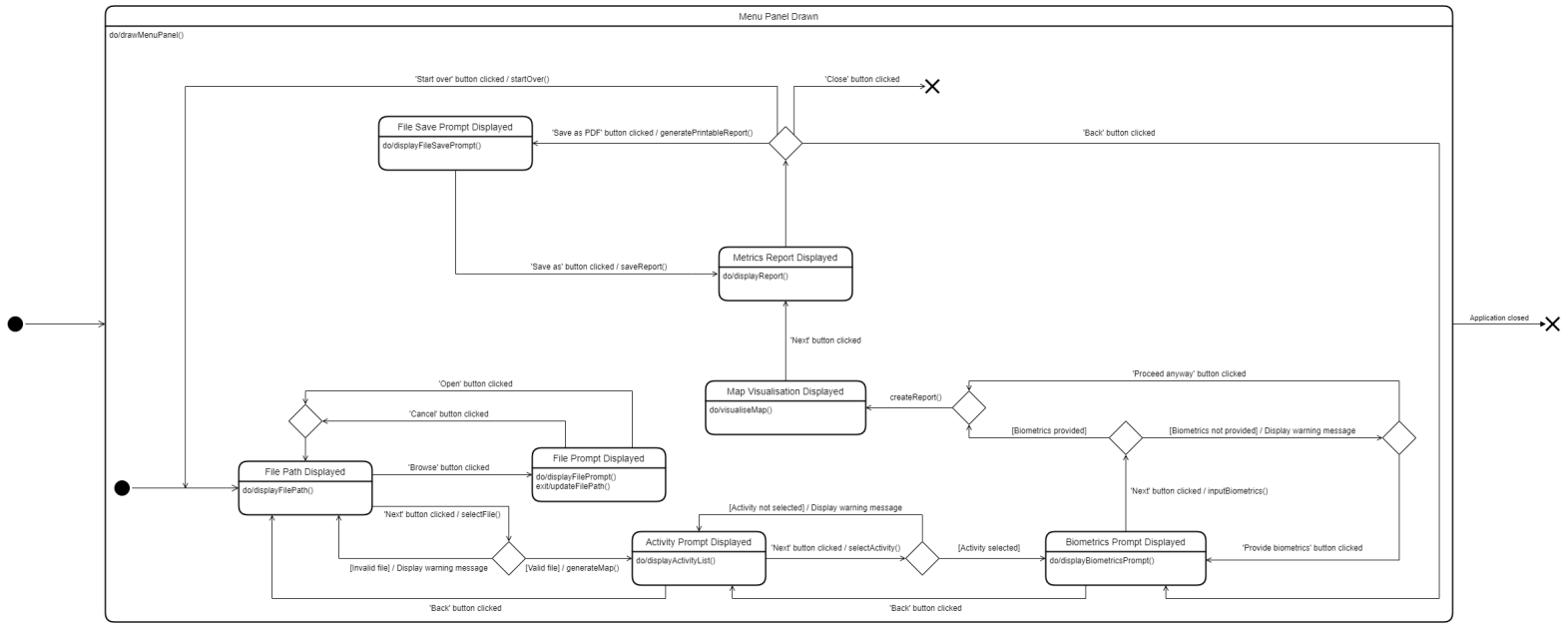
Implementation

Authors: Jeroen Klaver, Khagan Mammadov, David Breitling, Lucas Ponticelli

Changes in Java libraries from Assignment 1

- Added **JPX**: Our initial plan was to manually parse .gpx files, however we realised that this was unnecessary and using an external library would make our lives easier.
- Added **SLF4J-Log4j12**: A dependency of our **JavaFX** library, as it would function but print an error to `err` without a logger. Simply used to instantiate and configure a **Logger** object at the start of our application to hide the error message.
- Swapped **GMapsFX**: We initially attempted to visualise .gpx files with the help of the **GMapsFX** library, however we found the examples provided for the **GMapsFX** library to be lacking in detail and looked for alternatives instead. We found the **mapjfx** library in our search for alternatives and chose it as it provided a detailed demo project that we were able to utilise.
- Removed **JFoenix**: As stated in Assignment 1, we planned on using the **JFoenix** library to beautify our GUI if it provided a significant improvement over using plain **JavaFX**. We started the transition to **JFoenix** but then realised that we preferred the clear and simple design of our GUI and that the transition did not warrant the extra development time.

Our strategy for this assignment was to closely follow our previous diagrams while deviating when necessary. We made our diagrams as detailed as possible to get a good idea of the features we would later have to implement. This strategy paid off as we had not yet implemented the GUI for our MVP. The level of detail in our previous diagrams helped us visualise the changes we would have to make to our application as we transitioned from a CLI to GUI. More specifically, our GUI activity diagram visualised in great detail the parts of the GUI and their behaviour. This gave us a clear idea of what the interaction between the GUI and user of our application would look like. The GUI activity diagram can be seen below:



Listed below are the key features of our application and how we implemented them:

- .gpx file parsing: We used the **JPX** library to parse our .gpx files. An alternative to this would be to parse .gpx files manually but we ultimately decided it did not offer a significant advantage over using an external library.
- GUI: We used the **JavaFX** library to implement our GUI.
- Visualising .gpx files: We used the **mapjfx** library to visualise .gpx files. We used *mapjfx* in combination with **JavaFX** to display a visualisation of the chosen .gpx file. An alternative to this would be to use the **GMapsFX** library but we found the **mapjfx** library easier to use.
- Extensibility of activities: We implemented the abstract class **Activity** that is easily extensible with different activities.

Bonus

- Saving the report as a PDF: To satisfy our bonus requirement of “The tool should be able to generate a printable report that visualises and summarises in a nice way the map and the computed metrics” mentioned in Assignment 1, we used the **PDFBox** library to save generated reports as PDFs.
- Dynamically parsing the **activities** package: The main technical challenge of this project was to make our system “independent of the sports so to make it easily extensible”. We achieved this by implementing the abstract **Activity** class, but to make the process of adding new activities to our application as convenient as possible we avoided hardcoding the selection of available activities in our implementation. We achieved this by making use of the **Class** class and Factory design pattern to dynamically parse the **activities** package and instantiate the chosen activity by name. This means that new and existing activities in the **activities** package are automatically displayed to the user without the need to manually add them to any other part of our implementation.
- Factoring in elevation and user biometrics: We used the MET (Metabolic Equivalent of Task) formula to estimate the amount of calories burned. While the MET formula is a good estimate of calories burned, we wanted to provide a more accurate estimate by factoring in elevation, sex and age. As we calculate the metrics for each consecutive pair of waypoints, we are able to take into account the increases and decreases in elevation throughout the entire route.

In our source code, the main Java class needed for executing our system is located at:
src/main/java/softwaredesign/Main.java

The Jar file for directly executing our system can be found at:
out/artifacts/software-design-vu-2020.main.jar

The video showing the execution of our system can be found at:
https://youtu.be/VYLrIHdg6yM

Time logs

Team number	2			
Member	**Activity**	**Week number**	**Hours**	
Khagan, David	Implementation	5	2	1h each
Khagan, Lucas, Jeroen	Team meeting	5	3	1h each
Khagan, David	Implementation	6	4	2h each
Khagan	Design patterns	6	2	
Khagan, Jeroen	Class diagram	6	6	3h each
Khagan, Lucas, Jeroen	Object and state machine diagrams	7	9	3h each
David	Sequence diagrams	7	2	
Khagan, Lucas, Jeroen	Team meeting	7	3	1h each
Khagan, Lucas, Jeroen, David	Descriptions	7	16	4h each
Khagan	State machine and sequence diagrams	7	4	
Lucas	Proof reading	7	1	
Khagan	Implementation	7	2	
Khagan, Lucas, Jeroen	Descriptions and finishing touches	7	9	3h each
David	Finishing touches	7	1	
		TOTAL	64	